SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 5754

PROCJENA OČEKIVANOG STUPNJA POTPUNOSTI GENOMA IZ DOSTUPNIH OČITANJA

Matej Fureš

Zagreb, lipanj 2018

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA ZAVRŠNI RAD MODULA

Zagreb, 15. ožujka 2018.

ZAVRŠNI ZADATAK br. 5754

Pristupnik: **Matej Fureš (0036496166)**
Studij: Računarstvo
Modul: Računarska znanost

Zadatak: **Procjena očekivanog stupnja potpunosti genoma iz dostupnih očitanja**

Opis zadatka:

Sastavljanje genoma jedan je od najsloženijih zadataka u području bioinformatike i računalne biologije. Problem je NP težak i stoga je potrebno koristiti heurističke metode. Najveće probleme predstavljaju ponavljajuće regije. Ukoliko ne postoje očitanja koja mogu premostiti te regije ispravno sastavljanje nije moguće već je završni slijed fragmentiran. Cilj ovoga završnoga rada je, za poznati genom i zadana sekvencirana očitanja, utvrditi je li iz tih očitanja moguće sastaviti potpun genom.
Programski kod je potrebno komentirati i pri pisanju pratiti neki od standardnih stilova. Kompletnu aplikaciju postaviti na repozitorij Github.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 15. lipnja 2018.

Mentor:

Izv. prof. dr. sc. Milo Šikić

Predsjednik odbora za završni rad modula:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

1. Uvod

Sastavljanje genoma jedan je od najsloženijih zadataka u području bioinformatike i računalne biologije. Metode sastavljanja genoma (odnosno slijeda) iz dostupnih očitanja mogu se podijeliti na dva tipa: sastavljanje uz poznatu referencu te sastavljanje bez reference, tzv. *de novo* sastavljanje. Sastavljanje bez reference znatno je memorijski i vremenski iscrpnije od sastavljanja s poznatom referencom. Pri takvom sastavljanju bilo bi korisno imati alat koji bi unaprijed mogao reći ima li neki podskup uopće smisla za referencu, te bi se tako moglo znatno ubrzati vrijeme filtriranja. Ovaj završni rad bavi se izradom jednog takvog alata.

Alat je napisan u programskom jeziku C++ uz C++14 standard. Uz to su korišteni i neki drugi alati, kao što su *Minimap1* i *Gepard* za grafički prikaz rezultata.

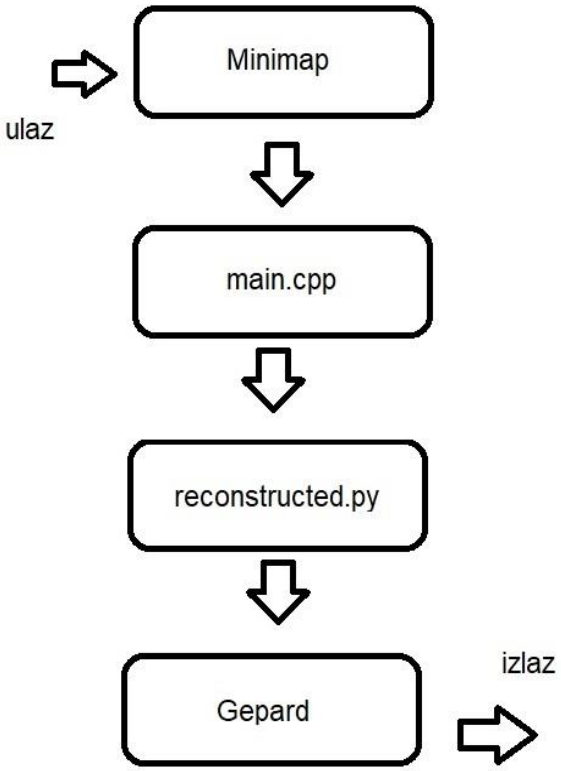
2. Algoritmi i strukture podataka

2.1 Algoritmi

Alat je zamišljen da radi na sljedeći način. Prvo se skup ulaznih podataka preda nekom postojećem alatu za mapiranje kako bi on mapirao zadani skup očitanja na referentni slijed. Mapiranja se zatim obrađuju unutar glavnog programa i svrstavaju u strukture pogodne za jednostavno baratanje našem alatu. Nakon što su mapiranja smještena u strukture, krećemo u postupak filtriranja podataka. Tijekom mapiranja u interesu nam je odstraniti mapiranja koja nemaju značaja za ciljani problem, npr. ona mapiranja koja su cijela sadržana u nekom drugom mapiranju.

Dodatno se filtriraju mapiranja i prema samom očitanju. Za svako pojedino očitanje ostavlja se samo jedno mapiranje, ono koje se ocjeni kao najbolje. Nakon što su preostala samo najbolja mapiranja, od istih gradimo graf. Na tom grafu pokušavamo napraviti obilazak, a iz tog obilaska zaključujemo može li se obaviti sastavljanje. Ako smo utvrdili da se sastavljanje može obaviti, preostaje nam grafički prikazati rezultat. Rezultat prikazujemo crtajući točkasti dijagram sastavljenog slijeda i originalne reference. Osim toga načina rada, postoji i način u kojemu predajemo dodatne podatke o preklapanjima koji služe kao još jedan uvjet za određivanje susjedstva između dva čvora čije se pozicije preklapaju.

Na slici 2.1 prikazane su sve jedinke i prijelazi od kojih se sastoji cjelokupan proces. Međupozivanje i preusmjeravanje obavlja *run.sh* skripta napisana u *bash*-u. Dvije *fasta* datoteke, od kojih jedna sadrži očitanja, a druga reference, šaljemo na ulaz *Minimap*-a koji te datoteke zahtijeva za određivanje mapiranja. Skripta tada poziva glavni program te mu predaje dvije datoteke *paf* formata. Glavni program tada filtrira, sastavlja graf i obilazi ga. Izlaz iz glavnog programa su mapiranja u *paf* formatu. U tom trenutku znamo koliko iznosi stupanj potpunosti genoma iz trenutnih očitanja te daljnji koraci nisu potrebni, već služe za grafički prikaz rezultata. Skripta tada predaje ta mapiranja *reconstruct.*py programu preko *stdin*-a. Tamo se sastavljaju sljedovi koje skripta, zajedno s referencom, prosljeđuje *Gepard*-u koji zatim crta točkasti dijagram.



Slika 2.1: Dijagram komponenti

2.2 Strukture podataka

Definicija osnovne strukture za gradnju grafa:

$$\{\, queryName, refStart, refEnd, qStart, qEnd, strand \,\}.$$

Struktura sadrži minimalan broj elemenata koji je potreban za izgradnju grafa i za izračunavanje duljine sastavljenog slijeda, odnosno dužine puta. Čvor čuva identifikator očitanja. Također čuva i pozicije početka i kraja mapiranja očitanja na referencu te poziciju početka i kraja na očitanju. Početak i kraj mapiranja na referenci kasnije koristimo kod preklapanja te pri određivanju susjedstva. Početak i kraj očitanja koristi se za rekonstrukcije duljine. U strukturi preostaje još samo podatak o polaritetu mapiranja. Taj podatak koristi se u algoritmu koji za određivanje susjedstva ne koristi samo pozicije mapiranja na reference, već i podatke koji dolaze kao izlaz nekog drugog programa, npr. *Minimap-a*.

Graf predstavlja mapiranja očitanja na neku konkretnu referencu. Sam graf je niz čvorova koji su uzlazno sortirani prema vrijednosti početka mapiranja na reference. Tako on ima svojstvo da se određivanje susjeda pojedinog čvora svodi na utvrđivanje ima li pojedini čvor manji *refStart* od *refEnd* zadanog čvora, i nalazi li se čvor s desne strane zadanom čvoru. Zbog izbacivanja mapiranja koja su jedna unutar drugih, nije moguće da dva čvora imaju jednak *refStart* ili *refEnd*, jer bi to povlačilo da je jedan sadržan u drugome, ili da su jednaki u objema varijablama. Filtracijom prema ocjeni mapiranja osigurano je da se pojedino očitanje može pojaviti samo jednom. Tako su izbjegnuti prebrojavanje pojedinih očitanja više puta i eksponencijalna složenost, no izgubljena je točnost. Filtracije omogućuju jednostavniji kod, te izbjegavanje složenosti koja uzrokuje dugotrajno izvođenje.

Uz to definirana je i struktura podataka koja definira izlat iz *Minimap*-a koja je objašnjena kasnije.

3. Izbacivanje suvišnih sadržanih mapiranja

Kod mapiranja često se dogodi da je neko mapiranje u potpunosti sadržano unutar drugoga. S obzirom da su takva mapiranja redundantna, povoljno je odstraniti ih odmah na početku programa te ga tako ubrzati. Problem otkrivanja je li neko mapiranje A sadržano unutar nekog čvora B svodi se na utvrđivanje istinitosti sljedećeg izraza:

$$(A.refStart >= B.refStart) \wedge (A.refEnd <= B.refEnd)$$

Rješavanje tog problema, odnosno izbacivanje suvišnih sadržanih mapiranja iz liste svih mapiranja, značajno utječe na brzinu izvođenja samog programa, te ga možemo riješiti na dva načina:

1.) <u>"Naivni" algoritam</u>

Za svako mapiranje A iz liste mapiranja i za svako drugo mapiranje B iz liste mapiranja, gdje vrijedi $A \neq B$, naivni algoritam provjerava vrijedi li gore dan izraz. Ako se utvrdi da je A sadržan unutar nekog drugog mapiranja, A se izbaci s liste mapiranja, a provjeravanje se nastavlja sve dok se ne provjere sva mapiranja.

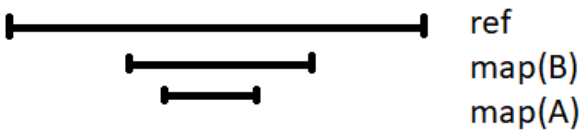
Vremenska složenost ovog "naivnog" algoritma je $O(N^2)$ te je zbog manje složenosti iskorišten sljedeći algoritam.

2.) <u>Algoritam *sweep line*</u>

Do boljeg rješenja moguće je doći primjenjujući algoritam *sweep line*. Kako bismo koristili taj algoritam, potrebni su pripremljeni podatci, odnosno sortirana lista mapiranja.

Listu prvo sortiramo uzlazno prema *refStart,* a zatim silazno prema *refEnd.* Upravo taj novi poredak omogućuje jednostavno rješavanje problema jer će se za istu početnu poziciju *refStart,,* ono mapiranje koje je duže, nalaziti na nižoj poziciji na listi, te nam je jednostavno odbaciti suvišna mapiranja. Algoritam se izvodi na sljedeći način: Prvo postavimo

vrijednost varijable B u trenutno najbolje mapiranje iz liste. Za svako mapiranje u listi A, krenuvši od drugog člana polja, provjeravamo vrijedi li $A.refEnd \leq B.refEnd,$ ako vrijedi, izbacujemo A iz algoritma, a u suprotnom postavljamo $B = A$ i nastavljamo prolaziti listom. Vremenska složenost samog *sweep_line* algoritma iznosi $O(N)$ gdje je N broj mapiranja. Budući da nam je za korištenje *sweep_line*-a potrebna sortirana lista mapiranja, u obzir moramo uzeti i složenost algoritma za sortiranje usporedbom, a ona iznosi $O(NlogN)$. Samim time, složenost ovog algoritma iznosi $O(NlogN)$



Slika 4.1: Primjer sadržanog mapiranja(A sadržan u B)

4. Filtriranja

4.1. Filtriranje mapiranja prema ocjeni mapiranja

Minimap često pojedino očitanje mapira na reference na više mjesta, samim time u sustavu je više potencijalnih mjesta na kojima bismo mogli iskoristiti pojedino mapiranje s kojim bi naš alat imao bolji rezultat. Na sva bi se ta pitanja moglo odgovoriti, ali zbog vremenske složenosti koju bi takva implementacija donijela sa sobom, traženje odgovora je beskorisno.

Da se izbjegne ta problematika i pojednostavni kod mapiranja pojedinog očitanja , ona se filtriraju prema ocjeni samog mapiranja, tako da ostane samo jedno mapiranje po očitanju u listi mapiranja. Ocjenu pojedinog mapiranja dobivamo preko kriterija.

Primjenjivi kriteriji za procjenu kvalitete mapiranja su broj baza koje se poklapaju u očitanju na reference te broj baza koje su prekrivene na referenci. Za obradu i dostavu tih podataka glavnom programu odgovoran je alata za mapiranje, u našem slučaju *Minimap.*

Konkretno filtriranje možemo riješiti u vremenskoj složenosti O(N), gdje je N broj očitanja, korištenjem hash tablice čiji su parovi:

$$(kljuc, vrijednost) \text{ jednaki } (ime, ocjenaMapiranja).$$

Za svako mapiranje treba pogledati trenutno stanje tablice. Moguća su tri slučaja:

1.) Tablica još nije postavljena. Na danu poziciju u tablici upisujemo trenutnu vrijednost
2.) Tablica je postavljena, ali trenutna ocjena tablice na zadanoj poziciji je manja od ocjene trenutne vrijednosti. Na danu poziciju u tablici upisujemo trenutnu vrijednost

3.) Tablica je postavljena, ali trenutna ocjena tablice na zadanoj pozicije je manja od ocjene trenutne vrijednosti. S obzirom na to da je već zabilježena bolja ocjena od trenutne, odbacujemo trenutnu vrijednost

Uz već prije spomenute prednosti korištenja ovakvog filtra, važno je istaknuti da zahvaljujući jedan na jedan odnosu između očitanja i mapiranja, koje ostvarujemo odmah na početku programa, dobivamo garanciju da nije moguće postojanje više mapiranja za jedno očitanje, pa ne moramo brinuti pri obilasku

4.2. Filtriranje preklapanja

Preklapanja su također filtrirana na način da između dva proizvoljna očitanja postoji najviše jedno preklapanje, a to će biti upravo ono sa najvećim brojem preklapajućih baza.

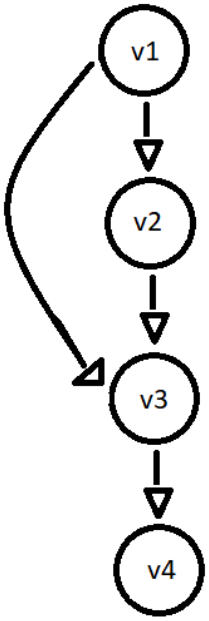
Uvjet da bi neko očitanje bilo sufiks, odnosno da bi bilo prefiks je jednostavan:

$$baze\_prije\_preklapanja(prefiks) > baze\_prije\_preklapanja(sufiks) \wedge baze\_nakon\_preklapanja(prefiks) < baze\_nakon\_preklapanja(sufiks)$$

Odnosno, broj baza na očitanju koje je prefiks mora biti veće od broja baza očitanja koje je sufiks prije samog preklapanja, dok nakon preklapanja mora vrijediti obrat, tj. broj baza na očitanju koje je prefiks mora biti manje od broja baza nakon preklapanja očitanja koje je sufiks.

Ako dani izraz ne vrijedi, zaključujemo da se ne radi o stvarnom preklapanju te to "preklapanje" odbacujemo. U suprotnom, dakle ako dani izraz vrijedi provjeravamo postoji li već neko preklapanje između danih očitanja te, ako ne postoji, postavljamo preklapanje između tih očitanja u trenutno preklapanje, a u suprotnom provjeravamo je li već postojeće očitanje lošije od trenutnog, te ga u tom slučaju mijenjamo sa trenutnim, dok za obrat odbacujemo trenutno preklapanje, tj. pamtimo bolje od dva preklapanja.

5. Algoritmi nad grafom



Slika 5.1: Usmjereni aciklički graf

Graf u našem alatu izgrađen je kao lista čvorova, a čvorovi su susjedi ako se preklapaju na reference. Formalno rečeno, brid usmjeren iz čvora A u čvor B postoji ako vrijedi sljedeći izraz:

$$B.refStart \leq A.refEnd$$

Taj izraz omogućuje da, sortiravši ulazne čvorove uzlazno po *refStart*, ne moramo stvarati ni matricu susjedstva ni listu susjedstva za svaki čvor, s obzirom na to da se svi potencijalni susjedi pojedinog čvora sigurno nalaze njemu s desne strane u listi, odnosno grafu.

Iterirajući listom od nekog čvora, sigurni smo da su mu potencijalni susjedi s desne strane, i kad nađemo prvi čvor koji je našem početnom čvoru s desne strane, a nije mu susjed, možemo zaključiti da niti jedan desno od tog čvora sigurno nije susjed početnom čvoru jer svi desno od prvog čvora koji nam nije susjed imaju od njega veći *refStart* te sigurno nisu susjedi našem početnom čvoru.

S obzirom na to kako je graf izgrađen i filtriran, on je usmjeren aciklički graf. Njegovim obilaskom želimo zaključiti mogu li se iz danih očitanja sastaviti. Za obilazak grafa koristimo BFS, odnosno algoritam za pretraživanje u širinu, koji pokrećemo na svakoj pojedinoj komponenti grafa, s obzirom na to da ih on može imati nekoliko.

Svakom čvoru se za vrijeme obilaska grafa pamte dvije informacije. Pamtimo je li određeni čvor već posjećen, a ako je posjećen, pamtimo i iz kojeg smo čvora ušli u njega. Te informacije čuvamo u mapi koja kao ključ prima poziciju čvora, a kao vrijednost vraća poziciju čvora iz kojeg se ušlo u naš čvor.

Na početku obilaska, sve vrijednosti svih čvorova u mapi postavljaju se na -1. Kad krenemo u obilazak, za svaki brid čitamo vrijednost čvora u koju mi želimo prijeći. Ako je ta vrijednost -1, onda znamo da je čvor još neposjećen te ga dodajemo u mapu za obradu, a kao vrijednost postavljamo poziciju čvora koji je pozvao prijelaz u naš čvor. Za slučaj da vrijednost nije -1, ništa se ne događa s obzirom na to da je čvor već dodan u mapu za obradu, a nismo ni prvi koji ga otvaraju.

Dodatno možemo i omogućiti da se dodatno provjerava jesu li čvorovi između kojih želimo napraviti prijelaz dobri susjedi. Pri normalnom obilasku grafa, ta funkcija, pri pokušavanju obavljanja prijelaza, uvijek vraća istinu, ali ako je mogućnost te provjere uključena, onda se pomoću podataka o preklapanju utvrđuje je li preklapanje tih čvorova ono koje smo zapamtili kao najbolje za ta dva očitanja, pa ako nije, funkcija vraća laž i ne obavlja se prijelaz, već se traži drugi mogući.

Nakon što smo uspjeli obići sve komponente grafa, rekonstruiramo put od čvora s najvećim *refEnd* koji pripada toj komponenti. Ti putevi će biti dio izlaza programa. Putevi su spremljeni u paf formatu s kojim radi *Minimap*. Pomoću tog izlaza konstruiramo sljedove u *FASTA* format i zatim ih uspoređujemo s referentnim sljedovima unutar programa *Gepard*.

Osim što se put rekonstruira, procjenjuje se i njegova duljina, odnosno duljina sastavljenog slijeda.

Duljinu puta računamo kao sumu duljina svih čvorova koji sudjeluju u sastavljanju slijed od koje oduzimamo sumu dužina svih preklapanja koja se pojavljuju na našem putu. Dužine preklapanja oduzimamo jer smo pri zbrajanju duljina čvorova dva puta računali na svaku dužinu preklapanja.

6. Dodatni alati

6.1 Minimap

Alat *Minimap1* je alat kojim u našem alatu mapiramo očitanja na referencu. *Minimap* je napravljen radi brzog i efikasnog mapiranja. Alat je otvorenog koda i izvorni kod dostupan je na *Githubu*. Mapiranja su opisana *PAF* (Pairwise mApping Format) formatom koji za svako mapiranje sadrži 12 kolona odvojenih znakom TAB. Kolone koje su nama zanimljive i iskorištene su sljedeće:

– Ime očitanja
– Duljina očitanja
– Početak mapiranja na očitanju
– Kraj mapiranja na očitanju
– Smjer, + ili –
– Ime reference na koju je mapirano očitanje
– Duljina reference
– Početak mapiranja na referenci
– Kraj mapiranja na referenci
– Broj istih baza na referenci i očitanju

6.2 Gepard

Alat *Gepard* (akronim od GEnome PAir - Rapid Dotter) je alat kojim grafički prikazujemo rezultate točkastim dijagramima (eng.) za neke dvije reference. Alat pronalazi riječi duljine 10 iz jedne reference u drugoj, osim ako nismo mijenjali postavke. Kako bi to napravio efikasno, za jednu referencu prvo izradi indeks (sufiksno polje) pa pretražuje riječ u logaritamskoj složenosti. Indeks je moguće spremiti pa crtanje traje kraće jer ne treba svaki put graditi indeks.

7. Rezultati

Prilikom testiranja alata korištene su 3 različite reference i više različitih skupova očitanja

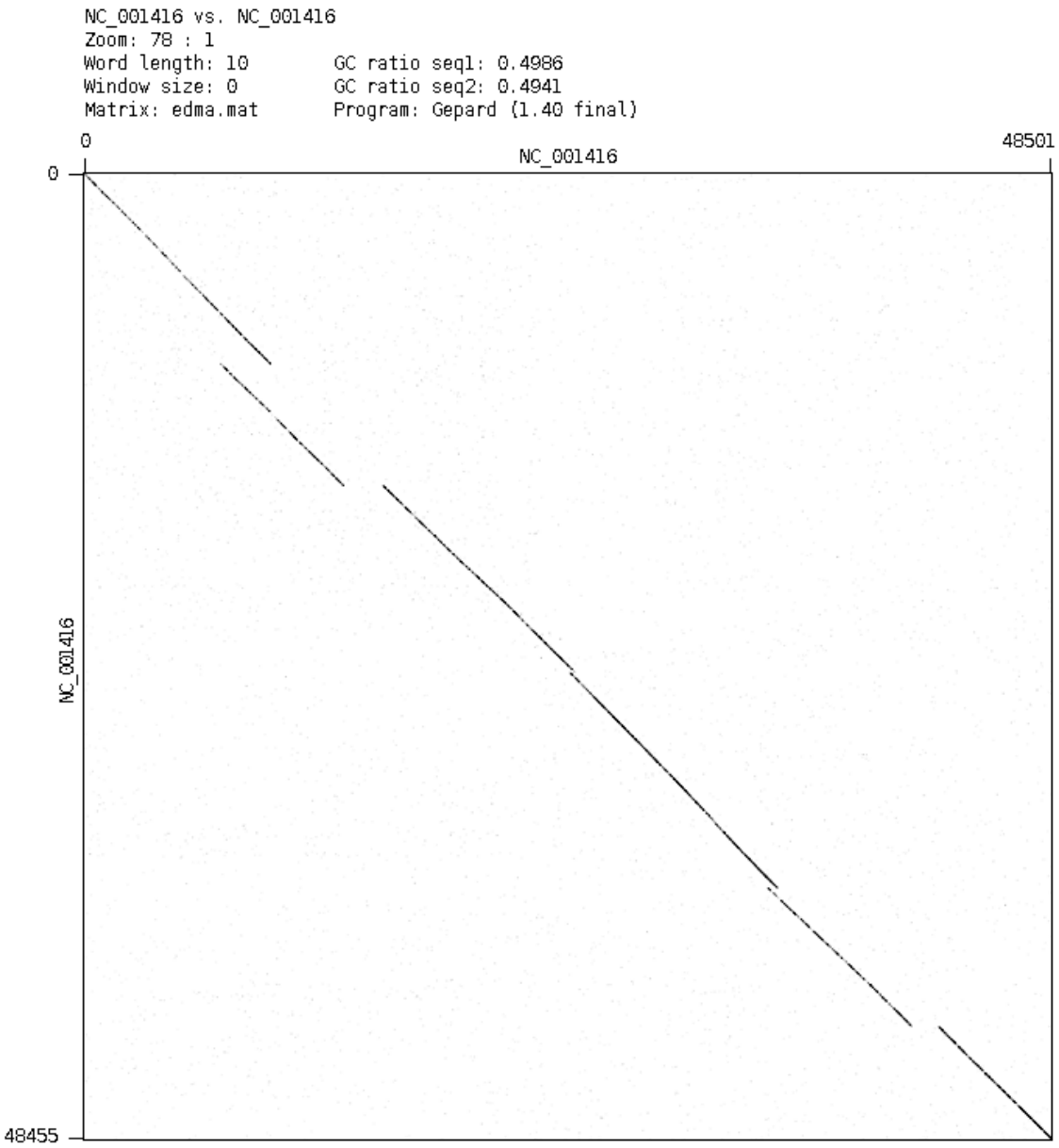
Tablica 7.1: Referenca

Naziv	*Duljina(base pair)*
Lambda	48502
E. Coli	4641652
Kvasac	12157105

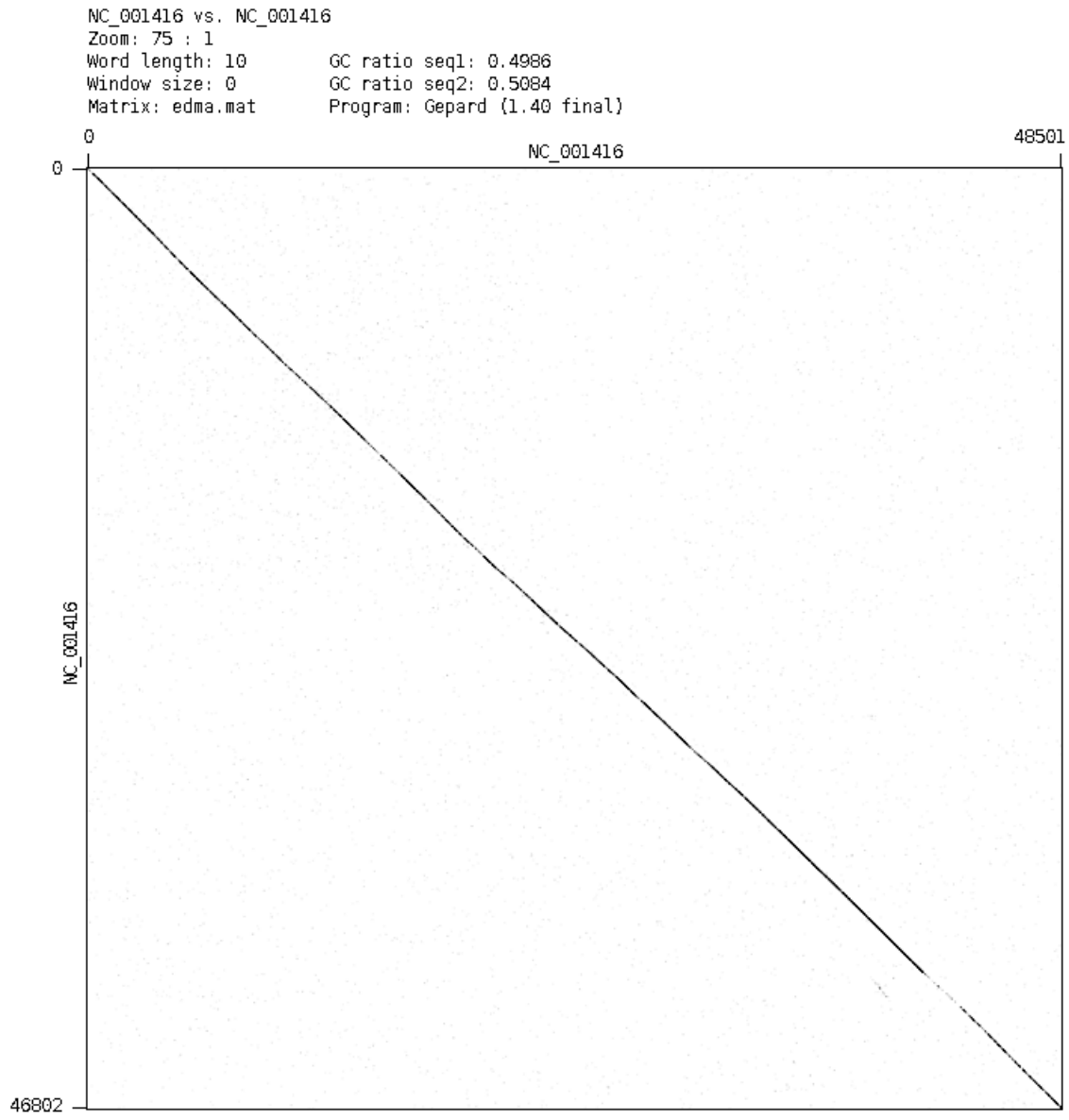
Prije nego što krenemo u detaljniju obradu podataka, bitno je napomenuti da je sastavljeni graf neprekinut za svaki slijed. Povezanost se ostvaruje preklapanjima na referenci, a eventuali promašaji na samoj referenci rezultat su *Minimap*-ovog algoritma mapiranja. Kao što je već rećeno, sastavljanje novog slijeda smatramo uspješnim ukoliko se svi omjeri duljina nalaze u intervalu $[0.9,1.1]$ odnosno, $\pm 10\%$. Omjeri koje promatramo su omjer duljine grafa sa duljinom reference, te omjer duljine grafa sa referencom pri kojem preskačemo susjede za koje prethodno nismo utvrdili da im je na tom mjestu najbolje preklapanje.

Tablica 7.2: Lambda

Očitanja	Omjer duljina	Omjer duljina s preklapanjem
R7	0.954703	0.997938
R9	0.954703	0.956373



Slika 7.1: Gepard izlaz lambde r7



Slika 7.2: Gepard izlaz lambde r9

Tablica 7.3: E. Coli

Očitanja	Omjer duljina	Omjer duljina s preklapanjem
R7	1.0229	1.02238

Slijed prikaz rezulta kvasca, njegovi rezultati su nešto zanimljiviji s obzirom da se radi o više kromosoma, ali rezultati su ponovno dobri budući da nemamo niti jedan omjer duljina manji od 0.9 odnosno veći od 1.1

Tablica 7.4: Rezultati kvasca

Kromosom	Omjer duljina	Omjer duljina s preklapanjima
NC_001133.9	0.9677	0.961563
NC_001134.8	0.976967	0.973983
NC_001135.5	0.962055	0.96227
NC_001136.10	0.982459	0.979625
NC_001137.3	0.976024	0.973755
NC_001138.5	0.977073	0.977306
NC_001139.9	0.974798	0.979904
NC_001140.6	0.973214	0.970711
NC_001141.2	0.98333	0.981602
NC_001142.9	0.971092	0.967741
NC_001143.9	0.971652	0.968264
NC_001144.5	0.976456	0.978271
NC_001145.3	0.98159	0.976051
NC_001146.8	0.974592	0.973124
NC_001147.6	0.981676	0.977802
NC_001148.4	0.972984	0.974059
NC_001224.1	0.987934	0.986955

Tablica 7.3: Karakteristike izvođenja skupa r7 nad referencom lambda

S obzirom da je omjer duljina na svakom testnom primjeru bio između 0.9 i 1.1 možemo zaključiti da je alat uspio složiti slijed u svakom testu.

Tablica 7.5: Vremenske i memorijske karakteristike izvođenja

Očitanje	Vrijeme izvođenja programa u sekundama	Maksimalno zauzeće memorije u kB
Lambda R7	0.02	3832
Lambda R9	0.16	6304
E. Coli R7	0.33	24084
Kvasac R9	54.6	1203084

8. Zaključak

Programski jezik *C++* i korištenje alata za mapiranje *Minimap* te alata za grafički prikaz točkastih dijagrama *Gepard* pokazalo se kao dobar izbor rješavanja problema utvrđivanja stupnja potpunosti genoma iz dostupnih očitanja. Alat Minimap sadži sve potrebne funkcionalnosti da nam pruži potrebne podatke za rješavanje zadanog problema.

Kako se i pretostavljalo prije početka samog rada, najkompleksniji dio rada bio je paziti na to da se izbjegnu velike vremenske složenosti unutar samog rješenja jer se radilo sa velikom količinom informacija. Dodatno je bilo potrebno nad podatcima, jer se uspostavilo da nam izlaz iz *Minimap*-a daje veliku količinu redundantnih informacija koja bi nam samo usporavala izvođenje.

Jednom kada smo uspjeli obraditi i filtrirati podatke koje nam je predao, krećemo u izgradnju grafa, koji se također pokazao kao dobro rješnje problema povezivanja preklapanja i određivanja puta. Nakon što smo stavili odgovarajuća preklapanja u listu, te tu listu sortirali, preostaje nam samo obaviti sam obilazak. S obzirom da naš graf ima više komponenti, obilazak je trebalo raditi iz svake pojedine komponente što nam je također usporilo samo vrijeme izvođenja programa.

Metode opisane u ovom završnom radu, pokazale su se dovoljno dobrima za utvrđivanje stupnja potpunosti genoma iz zadanih očitanja. Riješeni testovi dali su zadovoljavajuće rezultate, te bi se za daljnje unaprjeđenje ovog rješenja trebalo okrenuti konkretnijim zahtjevima i kompliciranijim organizmima.

Rješenju bi moglo doprinijeti zadavanje ulaznih datoteka u *paf* formatu umjesto slanja podataka na *Minimap*. Uz to, značajno bi pomoglo korištenje boljih alata za mapiranje, kvalitetnije filtracijske funkcije i eventualno omogućavanje više od jednog mapiranja po očitanju.

9. Literatura

[1] Jan Krumsiek, *Genome pair rapid dotter*, 19.2.2007, http://cube.univie.ac.at/gepard, 30.3.2018

[2] Heng Li, *Minimap* , 20.9.2017 https://github.com/lh3/minimap, 20.3.2018

[3] Heng Li, *Minimap2: pairwise alignment for nucleotide sequences, Bioinformatics*, bty191, https://doi.org/10.1093/bioinformatics/bty191

[4] *How to Install minimap in Ubuntu 16.04,* https://www.howtoinstall.me/ubuntu/16-04/minimap/ , 15.6.2018

[5] Završni rad, https://gitlab.com/Lifoc/zavrsniMatejFures2018

[6] Dropbox link sa primjerima za izvođenje implementacije, https://www.dropbox.com/sh/4tpi1z61f1kllm8/AADYs9PeOER6KMPYeiIANMgUa?dl=0

10. Sažetak

Procjena očekivanog stupnja potpunosti genoma iz dostupnih očitanja

Zadatak završnog rada alat je za provjeru mogućnosti sastavljanja genoma za dani skup očitavanja. Alat se radi u svrhu validacije skupa očitanja prilikom sastavljanja genoma te izačuna omjera duljina novo sastavljenog slijeda i početne reference. Problem je riješen korištenjem programskog jezika C++ te alata *Minimap* i *Gepard*. U samom programu korišteni su algoritmi *Sweep line* i *BFS* za filtraciju podataka, odnosno za obilazak grafa. Implementirano je rješenje zadovoljavajuće riješilo zadani problem, a njegova poboljšanja mogla bi se napraviti drugačijim parametrima pri ocjenjivanju kvalitete mapiranja ili korištenjem drugih alata.

Ključne riječi: C++, Minimap, Gepard, mapiranje, očitanje, graf, BFS, algoritam, Python, bash

11. Abstract

The Assesment of the Expected Genome Assembly Completness from Available Reads

The task of this assignment was implementation of a tool capable of testing the capability of assembling genomes from available reads. The tool was made for the purpose of validating sets of reads while assembling genomes and calculating ratios of lengths between original reference and newly made genome. The problem has been solved using programming language C++ and tools like Minimap and Gepard. In the solution we use Sweep line algorithm for data filtration and BFS algorithm for making a path on the graph. The implementation solved our problem to high degree, and eventual upgrades could be made by changing filtering methods or changing the tools that were used.

Key words: C++, Minimap, Gepard, maping, read, graph, BFS, algoritm, Python, bash